

tp

InetAddress

```
#pragma once

#include <arpa/inet.h>
#include <netinet/in.h>
#include <string>

// socket的地址协议类
class InetAddress
{
private:
    sockaddr_in addr_; // 表示地址协议的结构体。
public:
    InetAddress(const std::string &ip, uint16_t port); // 如果是监听的fd，用这个构造函数。
    InetAddress(const sockaddr_in &addr); // 如果是客户端连上来的fd，用这个构造函数。
    ~InetAddress();

    const char *ip() const; // 返回字符串表示的地址，例如：192.168.150.128
    uint16_t port() const; // 返回整数表示的端口，例如：80、8080
    const sockaddr *addr() const; // 返回addr_成员的地址，转换成了sockaddr。
};
```

The screenshot shows a terminal window with a file tree on the left and the code for `InetAddress.cpp` on the right. The file tree includes a project named `netserver` with subfolders `01`, `02`, and `03`. `03` contains files `client.cpp`, `InetAddress.cpp` (which is the current file), `InetAddress.h`, `makefile`, `tcpipoll.cpp`, and `tcpipoll.h`. The code in `InetAddress.cpp` is identical to the one shown in the previous code block.

```
> 打开的编辑器 1个未保存
PROJECT [SSH: 192.168.150.128]
> .vscode
> backup
< netserver
  > 01
  > 02
  < 03
    & client
    & client.cpp
    & InetAddress.cpp 2
    & InetAddress.h
    M makefile
    & tcpipoll
    & tcpipoll.cpp

netserver > 03 > & InetAddress.cpp > & InetAddress(const sockaddr_in)

1 #include "InetAddress.h"
2
3 /*
4 class InetAddress
5 {
6 private:
7     sockaddr_in addr_; // 表示地址协议的结构体。
8 public:
9     InetAddress(const std::string &ip, uint16_t port); // 如果是监听的fd，用这个构造函数。
10    InetAddress(const sockaddr_in &addr); // 如果是客户端连上来的fd，用这个构造函数。
11    ~InetAddress();
12
13    const char *ip() const; // 返回字符串表示的地址，例如：192.168.150.128
14    uint16_t port() const; // 返回整数表示的端口，例如：80、8080
15    const sockaddr *addr() const; // 返回addr_成员的地址，转换成了sockaddr。
16 };
17 */
18 InetAddress::InetAddress(const std::string &ip, uint16_t port) // 如果是监听的fd，用这个构造函数。
19 {
20     addr_.sin_family = AF_INET; // IPv4网络协议的套接字类型。
21     addr_.sin_addr.s_addr = inet_addr(ip.c_str()); // 服务端用于监听的ip地址。
22     addr_.sin_port = htons(port); // 服务端用于监听的端口。
23 }
24
25 InetAddress::InetAddress(const sockaddr_in &addr) // 如果是客户端连上来的fd，用这个构造函数。
```

```
InetAddress::InetAddress(const sockaddr_in addr):addr_(addr) // 如果是客户端连上来的fd，用这个构造函数。  
{  
}  
  
InetAddress::~InetAddress()  
{  
}  
  
const char *InetAddress::ip() const           // 返回字符串表示的地址，例如：192.168.150.128  
{  
}  
  
}  
  
uint16_t InetAddress::port() const           // 返回整数表示的端口，例如：80、8080  
{  
}  
  
}  
  
const sockaddr *InetAddress::addr() const    // 返回addr_成员的地址，转换成了sockaddr。  
{  
}  
|  
}  
}
```

```
server > 03 > C InetAddress.h > InetAddress  
1 #pragma once  
2  
3 #include <arpa/inet.h>  
4 #include <netinet/in.h>  
5 #include <string>  
6  
7 // socket的地址协议类  
8 class InetAddress  
9 {  
10 private:  
11     sockaddr_in addr_; // 表示地址协议的结构体。  
12 public:  
13     InetAddress(const std::string &ip,uint16_t port); // 如果是监听的fd，用这个构造函数。  
14     InetAddress(const sockaddr_in addr); // 如果是客户端连上来的fd，用这个构造函数。  
15     ~InetAddress();  
16  
17     const char *ip() const;           // 返回字符串表示的地址，例如：192.168.150.128  
18     uint16_t port() const;           // 返回整数表示的端口，例如：80、8080  
19     const sockaddr *addr() const;   // 返回addr_成员的地址，转换成了sockaddr。  
20 };
```

```
30 InetSocketAddress::~InetSocketAddress()
31 {
32 }
33 }
34
35 const char *InetSocketAddress::ip() const           // 返回字符串表示的地址，例如：192.168.150.128
36 {
37     return inet_ntoa(addr_.sin_addr);
38 }
39
40 uint16_t InetSocketAddress::port() const           // 返回整数表示的端口，例如：80、8080
41 {
42     return ntohs(addr_.sin_port);
43 }
44
45 const sockaddr *InetSocketAddress::addr() const    // 返回addr_成员的地址，转换成了sockaddr。
46 {
47     return (sockaddr*)&addr_;
48 }
```

```
#pragma once

#include <arpa/inet.h>
#include <netinet/in.h>
#include <string>

// socket的地址协议类
class InetAddress
{
private:
    sockaddr_in addr_; // 表示地址协议的结构体。
public:
    ~InetAddress();
    InetAddress(); // 如果是监听的fd，用这个构造函数。
    InetAddress(const std::string &ip, uint16_t port); // 如果是客户端连上来的fd，用这个构造函数。
    InetAddress(const sockaddr_in &addr); // 如果是客户端连上来的fd，用这个构造函数。
    void setaddr(sockaddr_in clientaddr); // 设置addr_成员的值。
};
```

```
InetAddress.cpp ✘  ●  tcpepoll.cpp ✘  ●  Socket.h  ●  SOCKET.cpp  ●  InetAddress.h
netserver > 04 > InetAddress.cpp > ...
1 #include "InetAddress.h"
2
3 InetAddress::InetAddress()
4 {
5
6 }
7
8 InetAddress::InetAddress(const std::string &ip, uint16_t port) // 如果是监听的fd，用这个构造函数。
9 {
10     addr_.sin_family = AF_INET; // IPv4网络协议的套接字类型。
11     addr_.sin_addr.s_addr = inet_addr(ip.c_str()); // 服务端用于监听的ip地址。
12     addr_.sin_port = htons(port); // 服务端用于监听的端口。
13 }
14
15 InetAddress::InetAddress(const sockaddr_in &addr):addr_(addr) // 如果是客户端连上来的fd，用这个构造函数。
16 {
17
18 }
19
20 InetAddress::~InetAddress()
21 {
22
23 }
24
25 const char *InetAddress::ip() const // 返回字符串表示的地址，例如：192.168.150.128
26 {
```

```
else if (evs[ii].events & (EPOLLIN|EPOLLPRI)) // 接收缓冲区中有数据可以读。
{
    if (evs[ii].data.fd==servsock.fd) // 如果是listenfd有事件，表示有新的客户端连上来。
    {
        /////////////////////////////////////////////////
        //sockaddr_in peeraddr;
        //socklen_t len = sizeof(peeraddr);
        //int clientfd = accept4(listenfd,(sockaddr*)&peeraddr,&len,SOCK_NONBLOCK);

        InetSocketAddress clientaddr; // 客户端的地址和协议。
        // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。
        // 还有，这里new出来的对象没有释放，这个问题以后再解决。
        Socket *clientsock=new Socket(servsock.accept(clientaddr));

        printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());

        // 为新客户端连接准备读事件，并添加到epoll中。
        ev.data.fd=clientsock->fd();
        ev.events=EPOLLIN|EPOLLET; // 边缘触发。
        epoll_ctl(epollfd,EPOLL_CTL_ADD,clientsock->fd(),&ev);
        ///////////////////////////////////////////////
    }
}
```

```
5 {
6     return ntohs(addr_.sin_port);
7 }
8
9 const sockaddr *InetAddress::addr() const // 返回addr_成员的地址，转换成了sockaddr。
10 {
11     return (sockaddr*)&addr_;
12 }
13
14 void InetAddress::setaddr(sockaddr_in clientaddr) // 设置addr_成员的值。
15 {
16     addr_=clientaddr;
17 }
```

Socket

> 打开的编辑器

PROJECT [SSH: 192.168.150.128]

- .vscode
- backup
- netserver
 - 01
 - 02
 - 03
 - 04
 - client
 - client.cpp
 - InetAddress.cpp
 - InetAddress.h
 - makefile
 - Socket.cpp
 - Socket.h
 - tcppepoll
 - tcppepoll.cpp

netserver > 04 > C Socket.h > Socket

```
1 #pragma once
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/tcp.h>
5 #include <string.h>
6 #include <errno.h>
7 #include <unistd.h>
8 #include "InetAddress.h"
9
10 // socket类。
11 class Socket
12 {
13 private:
14     const int fd_;           // Socket持有的fd，在构造函数中传进来。
15 public:
16     Socket(int fd);         // 构造函数，传入一个已准备好的fd。
17     ~Socket();              // 在析构函数中，将关闭fd_。
18
19     int fd() const;          // 返回fd_成员。
20     void setreuseaddr(bool on); // 设置SO_REUSEADDR选项，true-打开，false-关闭。
21     void setreuseport(bool on); // 设置SO_REUSEPORT选项。
22     void settcpnodelay(bool on); // 设置TCP_NODELAY选项。
23     void setkeepalive(bool on); // 设置SO_KEEPALIVE选项。
24     void bind(const InetAddress& servaddr); // 服务端的socket将调用此函数。
25     void listen(int nn=128); // 服务端的socket将调用此函数。
26     void accept(InetAddress& clientaddr); // 服务端的socket将调用此函数。
27 };
```

> 打开的编辑器 1个未保存

PROJECT [SSH: 192.168.150.128]

- .vscode
- backup
- netserver
 - 01
 - 02
 - 03
 - 04
 - client
 - client.cpp
 - InetAddress.cpp
 - InetAddress.h
 - makefile
 - Socket.cpp 1
 - Socket.h
 - tcppepoll
 - tcppepoll.cpp

netserver > 04 > C Socket.cpp > -Socket()

```
1 #include "Socket.h"
2
3 /*
4 // socket类。
5 class Socket
6 {
7 private:
8     const int fd_;           // Socket持有的fd，在构造函数中传进来。
9 public:
10    Socket(int fd);         // 构造函数，传入一个已准备好的fd。
11    ~Socket();              // 在析构函数中，将关闭fd_。
12
13    int fd() const;          // 返回fd_成员。
14    void setreuseaddr(bool on); // 设置SO_REUSEADDR选项，true-打开，false-关闭。
15    void setreuseport(bool on); // 设置SO_REUSEPORT选项。
16    void settcpnodelay(bool on); // 设置TCP_NODELAY选项。
17    void setkeepalive(bool on); // 设置SO_KEEPALIVE选项。
18    void bind(const InetAddress& servaddr); // 服务端的socket将调用此函数。
19    void listen(int nn=128); // 服务端的socket将调用此函数。
20    void accept(InetAddress& clientaddr); // 服务端的socket将调用此函数。
21 };
22 */
23
24 Socket::Socket(int fd); // 构造函数，传入一个已准备好的fd。
25 // 在析构函数中，将关闭fd_。
26 Socket::~Socket()
27 {
```

```
23
24 // 创建一个非阻塞的socket。
25 int createNonblocking()
26 {
27     // 创建服务端用于监听的listenfd。
28     int listenfd = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, IPPROTO_TCP);
29     if (listenfd < 0)
30     {
31         // perror("socket() failed"); exit(-1);
32         printf("%s:%s:%d listen socket create error:%d\n", __FILE__, __FUNCTION__, __LINE__, errno); exit(-1);
33     }
34     return listenfd;
35 }
36
37 // 构造函数，传入一个已准备好的fd。
38 Socket::Socket(int fd):fd_(fd)
39 {
40
41 }
```

```
42
43 // 构造函数，传入一个已准备好的fd。
44 Socket::Socket(int fd):fd_(fd)
45 {
46
47 }
48
49 // 在析构函数中，将关闭fd_。
50 Socket::~Socket()
51 {
52     ::close(fd_);
53 }
```

54 int Socket::fd() const // 返回fd_成员。

```
55 {
56     return fd_;
57 }
```

```
54 void Socket::settcpnodelay(bool on)
55 {
56     int optval = on ? 1 : 0;
57     ::setsockopt(fd_, IPPROTO_TCP, TCP_NODELAY, &optval, sizeof(optval)); // TCP_NODELAY包含头文件 <netinet/tcp.h>
58 }
59 #define SO_REUSEADDR 2
60 void Socket::setreuseaddr(bool on) 扩展到:
61 {
62     int optval = on ? 1 : 0; 2
63     ::setsockopt(fd_, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
64 }
65
66 void Socket::setreuseport(bool on)
67 {
68     int optval = on ? 1 : 0;
69     ::setsockopt(fd_, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof(optval));
70 }
71
72 void Socket::setkeepalive(bool on)
73 {
74     int optval = on ? 1 : 0;
75     ::setsockopt(fd_, SOL_SOCKET, SO_KEEPALIVE, &optval, sizeof(optval));
76 }
```

```
7 void Socket::bind(const InetAddress& servaddr)
8 {
9     if (::bind(fd_, servaddr.addr(), sizeof(sockaddr)) < 0 )
10    {
11        perror("bind() failed"); close(fd_); exit(-1);
12    }
13
14 void Socket::listen(int nn)
15 {
16     if (::listen(fd_,nn) != 0 )      // 在高并发的网络服务器中，第二个参数要大一些。
17     {
18         perror("listen() failed"); close(fd_); exit(-1);
19     }
20 }
```

```
21
22
23
24 void Socket::accept(InetAddress& clientaddr)
25 {
26     sockaddr_in peeraddr;
27     socklen_t len = sizeof(peeraddr);
28     int clientfd = accept4(fd_,(sockaddr*)&peeraddr,&len,SOCK_NONBLOCK);
29
30     InetAddress clientaddr(peeraddr);      // 客户端的地址和协议。
31
32 }
```

```

7 // socket的地址协议类
8 class InetAddress
9 {
10 private:
11     sockaddr_in addr_; // 表示地址协议的结构体。
12 public:
13     InetAddress(const std::string &ip,uint16_t port); // 如果是监听的fd，用这个构造函数。
14     InetAddress(const sockaddr_in addr); // 如果是客户端连上来的fd，用这个构造函数。
15     ~InetAddress();
16
17     const char *ip() const; // 返回字符串表示的地址，例如：192.168.150.128
18     uint16_t port() const; // 返回整数表示的端口，例如：80、8080
19     const sockaddr *addr() const; // 返回addr_成员的地址，转换成了sockaddr。
20     void setaddr(sockaddr_in clientaddr); // 设置addr_成员的值。
21 };

```

```

94     int Socket::accept(InetAddress& clientaddr)
95     {
96         sockaddr_in peeraddr;
97         socklen_t len = sizeof(peeraddr);
98         int clientfd = accept4(fd_,(sockaddr*)&peeraddr,&len,SOCK_NONBLOCK);
99
100        clientaddr.setaddr(peeraddr); // 客户端的地址和协议。
101
102        return clientfd;
103    }

```

Epoller

```

PROJECT [SSH: 192.168.150.128] 1 #pragma once
> .vscode 2 #include <stdio.h>
> backup 3 #include <stdlib.h>
< netserver 4 #include <errno.h>
> 01 5 #include <strings.h>
> 02 6 #include <string.h>
> 03 7 #include <sys/epoll.h>
> 04 8 #include <vector>
> 05 9 #include <unistd.h>
| client 10
| client.cpp 11 // Epoll类。
| Epoll.cpp 12 class Epoll
| Epoll.h 13 {
| InetAddress.cpp 14 private:
| InetAddress.h 15     static const int MaxEvents=100; // epoll_wait()返回事件数组的大小。
| makefile 16     int epollfd_=-1; // epoll句柄，在构造函数中创建。
| Socket.cpp 17     epoll_event events_[MaxEvents]; // 存放poll_wait()返回事件的数组，在构造函数中分配内存。
| Socket.h 18 public:
| tcpepoll 19     Epoll(); // 在构造函数中创建了epollfd_。
| tcpepoll.cpp 20     ~Epoll(); // 在析构函数中关闭epollfd_。
| 21
| 22     void addfd(int fd, uint32_t op); // 把fd和它需要监视的事件添加到红黑树上。
| 23     std::vector<epoll_event> loop(int timeout=-1); // 运行epoll_wait()，等待事件的发生，已发生的事件用vector容器返回。
| 24 }

```

```

ver > 05 > Epoll.cpp > Epoll::~Epoll()
#include "Epoll.h"

/*
// Epoll类。
class Epoll
{
private:
    static const int MaxEvents=100;           // epoll_wait()返回事件数组的大小。
    int epollfd_=-1;                         // epoll句柄，在构造函数中创建。
    epoll_event events_[MaxEvents];          // 存放poll_wait()返回事件的数组，在构造函数中分配内存。
public:
    Epoll();                                // 在构造函数中创建了epollfd_。
    ~Epoll();                               // 在析构函数中关闭epollfd_。

    void addfd(int fd, uint32_t op);          // 把fd和它需要监视的事件添加到红黑树上。
    std::vector<epoll_event> loop(int timeout=-1); // 运行epoll_wait()，等待事件的发生，已发生的事件用vector容器返回。
};*/
Epoll::Epoll()                           // 在构造函数中创建了epollfd_。
Epoll::~Epoll();                        // 在析构函数中关闭epollfd_。

```

```

Epoll::Epoll()                           // 在构造函数中创建了epollfd_。
{
    if ((epollfd_=epoll_create(1))==-1)    // 创建epoll句柄（红黑树）。
    {
        printf("epoll_create() failed(%d).\n",errno); exit(-1);
    }
}
Epoll::~Epoll()                          // 在析构函数中关闭epollfd_。
{
    close(epollfd_);
}

```

```

void Epoll::addfd(int fd, uint32_t op)           // 把fd和它需要监视的事件添加到红黑树上。
{
    epoll_event ev;   // 声明事件的数据结构。
    ev.data.fd=fd;    // 指定事件的自定义数据，会随着epoll_wait()返回的事件一并返回。
    ev.events=op;     // 让epoll监视fd的。

    if (epoll_ctl(epollfd_,EPOLL_CTL_ADD,fd,&ev)==-1) // 把需要监视的fd和它的事件加入epollfd中。
    {
        printf("epoll_ctl() failed(%d).\n",errno); exit(-1);
    }
}

std::vector<epoll_event> loop(int timeout) // 运行epoll_wait()，等待事件的发生，已发生的事件用vector容器返回。
{
    std::vector<epoll_event> evs;           // 存放epoll_wait()返回的事件。
}

```

```
42
43     std::vector<epoll_event> Epoll::loop(int timeout) // 运行epoll_wait(), 等待事件的发生, 已发生的事件用vector容器返回。
44 {
45     std::vector<epoll_event> evs;      // 存放epoll_wait()返回的事件。
46
47     bzero(events_, sizeof(events_));
48     int infds=epoll_wait(epollfd_,events_,MaxEvents,timeout);    // 等待监视的fd有事件发生。
49
50     // 返回失败。
51     if (infds < 0)
52     {
53         perror("epoll_wait() failed"); exit(-1);
54     }
55
56     // 超时。
57     if (infds == 0)
58     {
59         printf("epoll_wait() timeout.\n"); return;
60     }
61 }
```

```
netserver > 05 > Epoll.cpp > loop(int)
45     std::vector<epoll_event> evs;      // 存放epoll_wait()返回的事件。
46
47     bzero(events_, sizeof(events_));
48     int infds=epoll_wait(epollfd_,events_,MaxEvents,timeout);    // 等待监视的fd有事件发生。
49
50     // 返回失败。
51     if (infds < 0)
52     {
53         perror("epoll_wait() failed"); exit(-1);
54     }
55
56     // 超时。
57     if (infds == 0)
58     {
59         printf("epoll_wait() timeout.\n"); return evs;
60     }
61
62     // 如果infds>0, 表示有事件发生的fd的数量。
63     for (int ii=0;ii<infds;ii++)    // 遍历epoll返回的数组events_。
64     {
65         evs.push_back(events_[ii]);
66     }
67
68     return evs;
69 }
```

channel

PROJECT [SSH: 192.168.150.128]

```

3  /* 作者: 吴从周
4  */
5  #include <stdio.h>
6  #include <unistd.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <errno.h>
10 #include <sys/socket.h>
11 #include <sys/types.h>
12 #include <arpa/inet.h>
13 #include <sys/fcntl.h>
14 #include <sys/epoll.h>
15 #include <netinet/tcp.h> // TCP_NODELAY需要包含这个头文件。
16
17 class Channel
18 {
19 private:
20     int fd_;
21     bool islisten_=false; // true-监听的fd, false-客户端连上来的fd。
22     // .....更多的成员变量。
23 public:
24     Channel(int fd,bool islisten=false):fd_(fd),islisten_(islisten){}
25     int fd() { return fd_; }
26     bool islisten() { return islisten_; }
27     // .....更多的成员函数。
28 };
29

```

打开的编辑器 1个未保存 netserver > 06 > C Channel.h > Channel

```

PROJECT [SSH: 192.168.150.128]
1 #pragma once
2 #include <sys/epoll.h>
3 #include "Epoll.h"
4
5 class Epoll;
6
7 class Channel
8 {
9 private:
10     int fd_=-1; // Channel拥有的fd, Channel和fd是一对一的关系。
11     Epoll *ep_=nullptr; // Channel对应的红黑树, Channel与Epoll是多对一的关系, 一个Channel只对应一个Epoll。
12     bool inepoll_=false; // Channel是否已添加到epoll树上, 如果未添加, 调用epoll_ctl()的时候用EPOLL_CTL_ADD, 否则用EPOLL_CTL_MOD。
13     uint32_t events_=0; // fd 需要监视的事件。listenfd和clientfd需要监视EPOLLIN, clientfd还可能需要监视EPOLLOUT。
14     uint32_t revents_=0; // fd_已发生的事件。
15
16 public:
17     Channel(Epoll* ep,int fd); // 构造函数。
18     ~Channel(); // 析构函数。
19
20     int fd(); // 返回fd_成员。
21     void useet(); // 采用边缘触发。
22     void enablereading(); // 让epoll_wait()监视fd_的读事件。
23     void setinepoll(); // 把inepoll_成员的值设置为true。
24     void setevents(uint32_t ev); // 设置revents_成员的值为参数ev。
25     bool inpoll(); // 返回inepoll_成员。
26     uint32_t events(); // 返回events_成员。
27     uint32_t revents(); // 返回revents_成员。
28 };

```

资源管理器

打开的编辑器 1个未保存

PROJECT [SSH: 192.168.150.128]

> .vscode

> backup

< netserver

< 01

> demo

client

client.cpp

M makefile

tcpipoll

tcpipoll.cpp

> 02

> 03

> 04

> 05

< 06

Channel.cpp 2

Channel.h

client

client.cpp

Epoll.cpp

Epoll.h

InetAddress.cpp

InetAddress.h

M makefile

Socket.cpp

Socket.h

netserver > 06 > Channel.cpp > ...

```
18 void useet(); // 采用边缘触发。
19 void enablereading(); // 让epoll_wait()监视fd_的读事件。
20 void setinepoll(); // 把inepoll_成员的值设置为true。
21 void setevents(uint32_t ev); // 设置revents_成员的值为参数ev。
22 bool inpoll(); // 返回inepoll_成员。
23 uint32_t events(); // 返回events_成员。
24 uint32_t revents(); // 返回revents_成员。
25 }/*/
26
27 Channel::Channel(Epoll* ep,int fd):ep_(ep),fd_(fd) // 构造函数。
28 {
29
30 }
31
32 Channel::~Channel() // 析构函数。
33 {
34 // 在析构函数中，不要销毁ep_，也不能关闭fd_，因为这两个东西不属于Channel类，Channel类只是需要它们，使用它们而已。
35 }
36
37 int Channel::fd() // 返回fd_成员。
38 {
39     return fd_;
40 }
41
42 void Channel::useet() // 采用边缘触发。
43 {
44     events_=events|EPOLLET;
45 }
46
```

```
37 int Channel::fd() // 返回fd_成员。
38 {
39     return fd_;
40 }
41
42 void Channel::useet() // 采用边缘触发。
43 {
44     events_=events|EPOLLET;
45 }
```

```
56
57 void Channel::setevents(uint32_t ev) // 设置revents_成员的值为参数ev。
58 {
59     revents_=ev;
60 }
61
62 bool Channel::inepoll() // 返回inepoll_成员。
63 {
64     return inepoll_;
65 }
66
67 uint32_t Channel::events() // 返回events_成员。
68 {
69     return events_;
70 }
71
72 uint32_t Channel::revents() // 返回revents_成员。
73 {
74     return revents_;
75 }
```

```
netserver > 06 > C Epoll.h > ...
1 #pragma once
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <errno.h>
5 #include <strings.h>
6 #include <string.h>
7 #include <sys/epoll.h>
8 #include <vector>
9 #include <unistd.h>
10 #include "Channel.h"
11
12 class Channel;
13
14 // Epoll类。
15 class Epoll
16 {
17 private:
18     static const int MaxEvents=100;           // epoll_wait()返回事件数组的大小。
19     int epollfd_=-1;                         // epoll句柄，在构造函数中创建。
20     epoll_event events_[MaxEvents];         // 存放poll_wait()返回事件的数组，在构造函数中分配内存。
21 public:
22     Epoll();                                // 在构造函数中创建了epollfd_。
23     ~Epoll();                               // 在析构函数中关闭epollfd_。
24
25     // void addfd(int fd, uint32_t op);       // 把fd和它需要监视的事件添加到红黑树上。
26     void updatechannel(Channel *ch);          // 把channel添加/更新到红黑树上，channel中有fd，也有需要监视的事件。
27     std::vector<epoll_event> loop(int time); // 把c行。 // 运行epoll_wait(), 等待事件的发生，已发生的事件用vector容器返回。
28 };

```

```
// 把channel添加/更新到红黑树上，channel中有fd，也有需要监视的事件。
void Epoll::updatechannel(Channel *ch)
{
    epoll_event ev;                      // 声明事件的数据结构。
    ev.data.ptr=ch;                      // 指定channel。
    ev.events=ch->events();             // 指定事件。

    if (ch->inpoll())                 // 如果channel已经在树上了。
    {
        if (epoll_ctl(epollfd_,EPOLL_CTL_MOD,ch->fd(),&ev)==-1)
        {
            perror("epoll_ctl() failed.\n"); exit(-1);
        }
    }
    else                                // 如果channel不在树上。
    {
        if (epoll_ctl(epollfd_,EPOLL_CTL_ADD,ch->fd(),&ev)==-1)
        {
            perror("epoll_ctl() failed.\n"); exit(-1);
        }
        ch->setinpoll();               // 把channel的inpoll_成员设置为true。
    }
}
```

```
project [SSH: 192.168.1.25]
```

Channel.h Channel.cpp Epoll.h Epoll.cpp

```
netserver > 06 > Channel.cpp > enablereading()
```

```
34     // 在析构函数中，不要销毁ep_，也不能关闭fd_，因为这两个东西不属于Channel类，Channel类只
35 }
36
37 int Channel::fd() // 返回fd_成员。
38 {
39     return fd_;
40 }
41
42 void Channel::useet() // 采用边缘触发。
43 {
44     events_=events_|EPOLLET;
45 }
46
47 void Channel::enablereading() // 让epoll_wait()监视fd_的读事件。
48 {
49     events_|=EPOLLIN;
50     ep_->updatechannel(this);
51 }
52
53 void Channel::setinepoll() // 把inepoll_成员的值设置为true。
54 {
55     inepoll_=true;
56 }
57
58 void Channel::setrevents(uint32_t ev) // 设置revents_成员的值为参数ev。
59 }
```

```
Channel.h Channel.cpp Epoll.h Epoll.cpp
```

```
netserver > 06 > Epoll.h > Epoll > loop(int)
```

```
8 #include <vector>
9 #include <unistd.h>
10 #include "Channel.h"
11
12 class Channel;
13
14 // Epoll类。
15 class Epoll
16 {
17 private:
18     static const int MaxEvents=100; // epoll_wait()返回事件数组的大小。
19     int epollfd_=-1; // epoll句柄，在构造函数中创建。
20     epoll_event events_[MaxEvents]; // 存放poll_wait()返回事件的数组，在构造函数中分配内存。
21 public:
22     Epoll(); // 在构造函数中创建了epollfd_。
23     ~Epoll(); // 在析构函数中关闭epollfd_。
24
25     // void addfd(int fd, uint32_t op); // 把fd和它需要监视的事件添加到红黑树上。
26     void updatechannel(Channel *ch); // 把channel添加/更新到红黑树上，channel中有fd，也有需要监视的事件。
27     // std::vector<epoll_event> loop(int timeout=-1); // 运行epoll_wait()，等待事件的发生，已发生的事件用vector容器返回。
28     std::vector<Channel *> loop(int timeout=-1); // 运行epoll_wait()，等待事件的发生，已发生的事件用vector容器返回。
29 };
```

```
// 运行epoll_wait(), 等待事件的发生, 已发生的事件用vector容器返回。
std::vector<Channel *> Epoll::loop(int timeout)
{
    std::vector<Channel *> channels; // 存放epoll_wait()返回的事件。

    bzero(events_, sizeof(events_));
    int infds=epoll_wait(epollfd_,events_,MaxEvents,timeout); // 等待监视的fd有事件发生。
```

```
// 返回失败。
if (infds < 0)
{
    perror("epoll_wait() failed"); exit(-1);
}
```

```
110 // 返回失败。
111 if (infds < 0)
112 {
113     perror("epoll_wait() failed"); exit(-1);
114 }
115
116 // 超时。
117 if (infds == 0)
118 {
119     printf("epoll_wait() timeout.\n");
120 }
121
122 // 如果infds>0, 表示有事件发生的fd的数量。
123 for (int ii=0;ii<infds;ii++) // 遍历epoll返回的数组events_。
124 {
125     // evs.push_back(events_[ii]);
126     Channel *ch=(Channel *)events_[ii].data.ptr; // 取出已发生事件的channel。
127     ch->setevents(events_[ii].events); // 设置channel的revents_成员。
128     channels.push_back(ch);
129 }
130
131 return channels;
132 }
```

```
133 }
```

```

15
16 public:
17     Channel(Epoll* ep,int fd);    // 构造函数。
18     ~Channel();                  // 析构函数。
19
20     int fd();                   // 返回fd_成员。
21     void useet();               // 采用边缘触发。
22     void enablereading();      // 让epoll_wait()监视fd_的读事件。
23     void setinepoll();          // 把inepoll_成员的值设置为true。
24     void setevents(uint32_t ev); // 设置revents_成员的值为参数ev。
25     bool inpoll();              // 返回inepoll_成员。
26     uint32_t events();          // 返回events_成员。
27     uint32_t revents();         // 返回revents_成员。
28
29     void handleevent();        // 事件处理函数， epoll_wait()返回的时候，执行它。
30 };

```

Channel.h

```

6
7 class Channel
8 {
9 private:
10     int fd_=-1;                // Channel拥有的fd， Channel和fd是一对一的关系。
11     Epoll *ep_=nullptr;        // Channel对应的红黑树， Channel与Epoll是多对一的关系，一个Channel只对应一个Epoll。
12     bool inepoll_=false;       // Channel是否已添加到epoll树上，如果未添加，调用epoll_ctl()的时候用EPOLL_CTL_ADD，否则用EPOL
13     uint32_t events_=0;         // fd_需要监视的事件。listenfd和clientfd需要监视EPOLLIN， clientfd还可能需要监视EPOLLOUT。
14     uint32_t revents_=0;        // fd_已发生的事情。
15     bool islisten_=false;       // listenfd取值为true，客户端连上来的fd取值false。
16
17 public:
18     Channel(Epoll* ep,int fd,bool islisten); // 构造函数。
19     ~Channel();                  // 析构函数。
20
21     int fd();                   // 返回fd_成员。
22     void useet();               // 采用边缘触发。
23     void enablereading();      // 让epoll_wait()监视fd_的读事件。
24     void setinepoll();          // 把inepoll_成员的值设置为true。
25     void setevents(uint32_t ev); // 设置revents_成员的值为参数ev。
26     bool inpoll();              // 返回inepoll_成员。
27     uint32_t events();          // 返回events_成员。
28     uint32_t revents();         // 返回revents_成员。
29
30     void handleevent();        // 事件处理函数， epoll_wait()返回的时候，执行它。
31 };

```

```

26
27     Channel::Channel(Epoll* ep,int fd,bool islisten):ep_(ep),fd_(fd),islisten_(islisten) // 构造函数。
28 {
29
30 }

```

```

1 #pragma once
2 #include <sys/epoll.h>
3 #include "Epoll.h"
4 #include "InetAddress.h"
5 #include "Socket.h"
6
7 class Epoll;
8
9 class Channel {
10 {
11 private:
12     int fd = -1; // Channel拥有的fd, Channel和fd是一对一的关系。
13     Epoll *ep = nullptr; // Channel对应的红黑树, Channel与Epoll是多对一的关系, 一个Channel只对应一个Epoll。
14     bool inepoll = false; // Channel是否已添加到epoll树上, 如果未添加, 调用epoll_ctl()的时候用EPOLL_CTL_ADD, 否则用EPOL
15     uint32_t events = 0; // fd_需要监视的事件。listenfd和clientfd需要监视EPOLLIN, clientfd还可能需要监视EPOLLOUT。
16     uint32_t revents = 0; // fd_已发生的事件。
17     bool islisten = false; // listenfd取值为true, 客户端连上来的fd取值false。
18
19 public:
20     Channel(Epoll* ep, int fd, bool islisten); // 构造函数。
21     ~Channel(); // 析构函数。
22 }
```

```

C Channel.h X C Channel.cpp 5 G tcpepoll.cpp
netserver > 07 > C Channel.h > Channel > handleevent(Socket *)
19 public:
20     Channel(Epoll* ep, int fd, bool islisten); // 构造函数。
21     ~Channel(); // 析构函数。
22
23     int fd(); // 返回fd_成员。
24     void useet(); // 采用边缘触发。
25     void enablereading(); // 让epoll_wait()监视fd_的读事件。
26     void setinepoll(); // 把inepoll_成员的值设置为true。
27     void setrevents(uint32_t ev); // 设置revents_成员的值为参数ev。
28     bool inpoll(); // 返回inepoll_成员。
29     uint32_t events(); // 返回events_成员。
30     uint32_t revents(); // 返回revents_成员。
31
32     void handleevent(Socket *servsock); // 事件处理函数, epoll_wait()返回的时候, 执行它。

```

文件: Channel.h

```
netserver > 07 > Channel.cpp > handleevent(Socket *)
```

```
77
78 // 事件处理函数, epoll_wait()返回的时候, 执行它。
79 void Channel::handleevent(Socket *servsock)
80 {
81     if (revents_ & EPOLLRDHUP)           // 对方已关闭, 有些系统检测不到, 可以使用EPOLLIN, recv()返回0。
82     {
83         printf("client(eventfd=%d) disconnected.\n", fd_);
84         close(fd_);          // 关闭客户端的fd。
85     }                                // 普通数据 带外数据
86     else if (revents_ & (EPOLLIN|EPOLLPRI)) // 接收缓冲区中有数据可以读。
87     {
88         if (islisten_==true) // 如果是listenfd有事件, 表示有新的客户端连上来。
89         {
90             /////////////////////////////////
91             InetSocketAddress clientaddr; // 客户端的地址和协议。
92             // 注意, clientsock只能new出来, 不能在栈上, 否则析构函数会关闭fd。
93             // 还有, 这里new出来的对象没有释放, 这个问题以后再解决。
94             Socket *clientsock=new Socket(servsock->accept(clientaddr));
95
96             printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
97
98             // 为新客户端连接准备读事件, 并添加到epoll中。
99             Channel *clientchannel=new Channel(&ep,clientsock->fd()); // 这里new出来的对象没有释放, 这个问题以后再解决。
100            clientchannel->useet(); // 客户端连上来的fd采用边缘触发。
101            clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件。
102        }                                ///////////////////////////////
103    }
```

```
86     else if (revents_ & (EPOLLIN|EPOLLPRI)) // 接收缓冲区中有数据可以读。
87     {
88         if (islisten_==true) // 如果是listenfd有事件, 表示有新的客户端连上来。
89         {
90             /////////////////////////////////
91             InetSocketAddress clientaddr; // 客户端的地址和协议。
92             // 注意, clientsock只能new出来, 不能在栈上, 否则析构函数会关闭fd。
93             // 还有, 这里new出来的对象没有释放, 这个问题以后再解决。
94             Socket *clientsock=new Socket(servsock->accept(clientaddr));
95
96             printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
97
98             // 为新客户端连接准备读事件, 并添加到epoll中。
99             Channel *clientchannel=new Channel(&ep,clientsock->fd(),false); // 这里new出来的对象没有释放, 这个问题以后再解决。
100            clientchannel->useet(); // 客户端连上来的fd采用边缘触发。
101            clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件。
102        }                                ///////////////////////////////
103    }
104    else // 如果是客户端连接的fd有事件。
105    {
106        ///////////////////////////////
107        char buffer[1024];
108        while (true) // 由于使用非阻塞IO, 一次读取buffer大小数据, 直到全部的数据读取完毕。
109        {
```

```
104     else // 如果是客户端连接的fd有事件。
105     {
106         ///////////////////////////////////////////////////////////////////
107         char buffer[1024];
108         while (true) // 由于使用非阻塞IO，一次读取buffer大小数据，直到全部的数据读取完毕。
109         {
110             bzero(&buffer, sizeof(buffer));
111             ssize_t nread = read(fd_, buffer, sizeof(buffer));
112             if (nread > 0) // 成功的读取到了数据。
113             {
114                 // 把接收到的报文内容原封不动的发回去。
115                 printf("recv(eventfd=%d):%s\n", fd_, buffer);
116                 send(fd_, buffer, strlen(buffer), 0);
117             }
118             else if (nread == -1 && errno == EINTR) // 读取数据的时候被信号中断，继续读取。
119             {
120                 continue;
121             }
122             else if (nread == -1 && ((errno == EAGAIN) || (errno == EWOULDBLOCK))) // 全部的数据已读取完毕。
123             {
124                 break;
125             }
126             else if (nread == 0) // 客户端连接已断开。
127             {
128                 printf("client(eventfd=%d) disconnected.\n", fd_);
129                 close(fd_); // 关闭客户端的fd。
130                 break;
131             }
132         }
133     }
```

3.81x 超清 字幕 查找

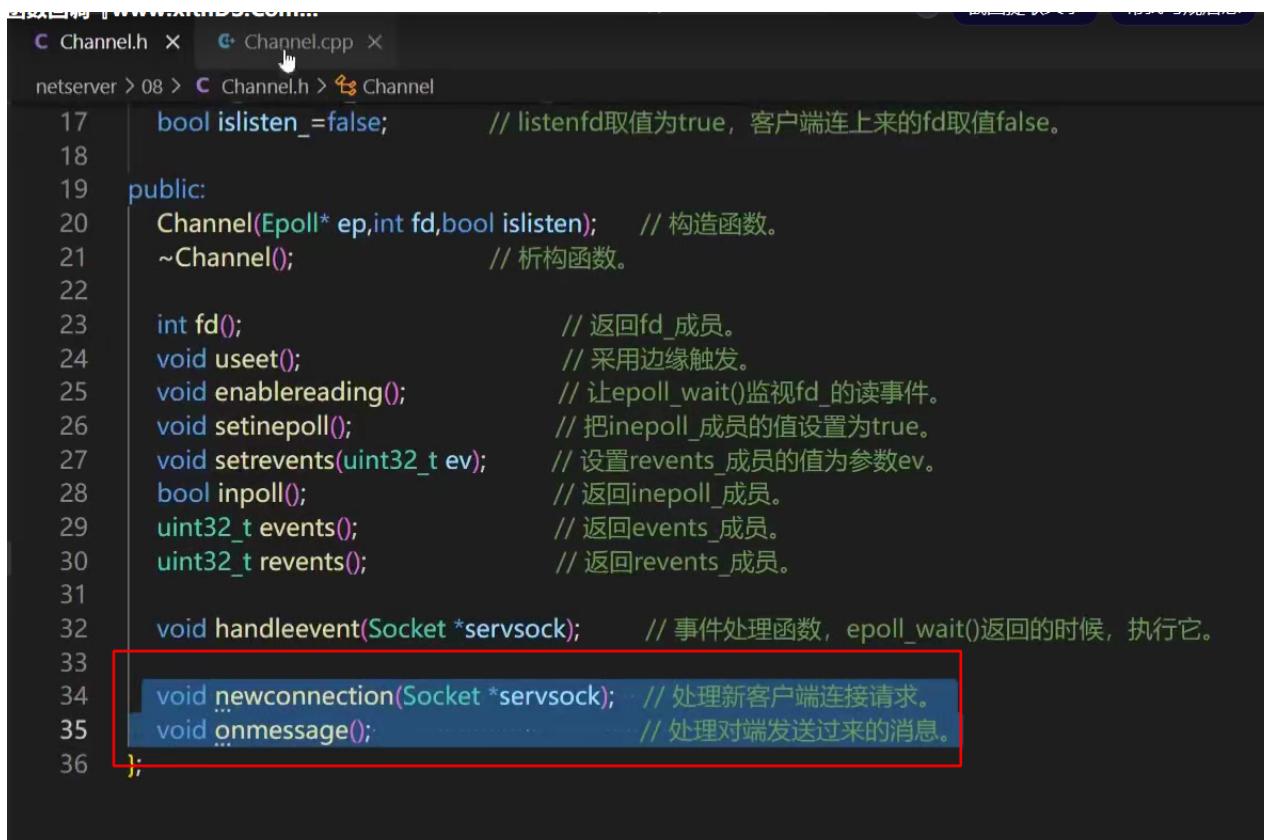
```
netserver > 07 > Channel.cpp > handleEvent(Socket *)
116         send(fd_, buffer, strlen(buffer), 0);
117     }
118     else if (nread == -1 && errno == EINTR) // 读取数据的时候被信号中断，继续读取。
119     {
120         continue;
121     }
122     else if (nread == -1 && ((errno == EAGAIN) || (errno == EWOULDBLOCK))) // 全部的数据已读取完毕。
123     {
124         break;
125     }
126     else if (nread == 0) // 客户端连接已断开。
127     {
128         printf("client(eventfd=%d) disconnected.\n", fd_);
129         close(fd_); // 关闭客户端的fd。
130         break;
131     }
132 }
133 }
134 }
135 else if (revents_ & EPOLLOUT) // 有数据需要写，暂时没有代码，以后再说。
136 {
137 }
138 else // 其它事件，都视为错误。
139 {
140     printf("client(eventfd=%d) error.\n", fd_);
141     close(fd_); // 关闭客户端的fd。
142 }
```

```

28] // 事件处理函数, epoll_wait()返回的时候, 执行它。
79 void Channel::handleevent(Socket *servsock)
80 {
81     if (revents_ & EPOLLRDHUP)           // 对方已关闭, 有些系统检测不到, 可以使用EPOLLIN, recv()返回0。
82     {
83         printf("client(eventfd=%d) disconnected.\n", fd_);
84         close(fd_);                  // 关闭客户端的fd。
85     }                                // 普通数据 带外数据
86     else if (revents_ & (EPOLLIN|EPOLLPRI)) // 接收缓冲区中有数据可以读。
87     {
88         if (islisten_ == true) // 如果是listenfd有事件, 表示有新的客户端连上来。
89         {
90             /////////////////////////////////////////////////
91             InetSocketAddress clientaddr; // 客户端的地址和协议。
92             // 注意, clientsock只能new出来, 不能在线上, 否则析构函数会关闭fd。
93             // 还有, 这里new出来的对象没有释放, 这个问题以后再解决。
94             Socket *clientsock = new Socket(servsock->accept(clientaddr));
95
96             printf("accept client(fd=%d,ip=%s,port=%d) ok.\n", clientsock->fd(), clientaddr.ip(), clientaddr.port());
97
98             // 为新客户端连接准备读事件, 并添加到epoll中。
99             Channel *clientchannel = new Channel(ep_, clientsock->fd(), false); // 这里new出来的对象没有释放, 这个问题以后再解决。
100            clientchannel->useet(); // 客户端连上来的fd采用边缘触发。
101            clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件。
102        }                                ///////////////////////////////////////////////////
103    }

```

处理回调



```

C Channel.h X  C Channel.cpp X
netserver > 08 > C Channel.h > Channel
17     bool islisten_=false;           // listenfd取值为true, 客户端连上来的fd取值false。
18
19 public:
20     Channel(Epoll* ep,int fd,bool islisten); // 构造函数。
21     ~Channel();                         // 析构函数。
22
23     int fd();                          // 返回fd_成员。
24     void useet();                     // 采用边缘触发。
25     void enablereading();            // 让epoll_wait()监视fd_的读事件。
26     void setinepoll();               // 把inepoll_成员的值设置为true。
27     void setrevents(uint32_t ev);    // 设置revents_成员的值为参数ev。
28     bool inpoll();                  // 返回inepoll_成员。
29     uint32_t events();              // 返回events_成员。
30     uint32_t revents();             // 返回revents_成员。
31
32     void handleevent(Socket *servsock); // 事件处理函数, epoll_wait()返回的时候, 执行它。
33
34     void newconnection(Socket *servsock); // 处理新客户端连接请求。
35     void onmessage();                 // 处理对端发送过来的消息。
36 }

```

```
20
21 // 处理新客户端连接请求。
22 void Channel::newconnection(Socket *servsock)
23 {
24     InetSocketAddress clientaddr; // 客户端的地址和协议。
25     // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。
26     // 还有，这里new出来的对象没有释放，这个问题以后再解决。
27     Socket *clientsock=new Socket(servsock->accept(clientaddr));
28
29     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
30
31     // 为新客户端连接准备读事件，并添加到epoll中。
32     Channel *clientchannel=new Channel(ep_,clientsock->fd(),false); // 这里new出来的对象没有释放，这个问题以后再解决。
33     clientchannel->useet(); // 客户端连上来的fd采用边缘触发。
34     clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
35 }
36
```

```
netserver > 08 > Channel.cpp > onmessage()
138 void Channel::onmessage()
139 {
140     char buffer[1024];
141     while (true) // 由于使用非阻塞IO，一次读取buffer大小数据，直到全部的数据读取完毕。
142     {
143         bzero(&buffer, sizeof(buffer));
144         ssize_t nread = read(fd_, buffer, sizeof(buffer));
145         if (nread > 0) // 成功的读取到了数据。
146         {
147             // 把接收到的报文内容原封不动的发回去。
148             printf("recv(eventfd=%d):%s\n",fd_,buffer);
149             send(fd_,buffer,strlen(buffer),0);
150         }
151         else if (nread == -1 && errno == EINTR) // 读取数据的时候被信号中断，继续读取。
152         {
153             continue;
154         }
155         else if (nread == -1 && ((errno == EAGAIN) || (errno == EWOULDBLOCK))) // 全部的数据已读取完毕。
156     }
157     break;
158 }
159 else if (nread == 0) // 客户端连接已断开。
160 {
161     printf("client(eventfd=%d) disconnected.\n",fd_);
162     close(fd_); // 关闭客户端的fd。
163     break;
164 }
```

3.81x 超清 字幕 查找

```
netserver > 08 > C Channel.h > Channel
15     bool inepoll_=false;           // Channel是否已添加到epoll树上，如果未添加，调用epoll_ctl()的时候用EPOL
16     uint32_t events_=0;            // fd_需要监视的事件。listenfd和clientfd需要监视EPOLLIN，clientfd还可能需
17     uint32_t revents_=0;           // fd_已发生的事件。
18     bool islisten_=false;          // listenfd取值为true，客户端连上来的fd取值false。
19     std::function<void()> readcallback_; // fd_读事件的回调函数。
20
21 public:
22     Channel(Epoll* ep,int fd,bool islisten); // 构造函数。
23     ~Channel();                         // 析构函数。
24
25     int fd();                          // 返回fd_成员。
26     void useet();                     // 采用边缘触发。
27     void enablereading();             // 让epoll_wait()监视fd_的读事件。
28     void setinepoll();                // 把inepoll_成员的值设置为true。
29     void setrevents(uint32_t ev);      // 设置revents_成员的值为参数ev。
30     bool inpoll();                   // 返回inepoll_成员。
31     uint32_t events();                // 返回events_成员。
32     uint32_t revents();               // 返回revents_成员。
33
34     void handleevent(Socket *servsock); // 事件处理函数，epoll_wait()返回的时候，执行它。
35
36     void newconnection(Socket *servsock); // 处理新客户端连接请求。
37     void onmessage();                  // 处理对端发送过来的消息。
38     void setreadcallback(std::function<void()> fn); // 设置fd_读事件的回调函数。
39 }
```

```
5 // 设置fd_读事件的回调函数。
6 void Channel::setreadcallback(std::function<void()> fn)
7 {
8     readcallback_=fn;
9 }
0 }
```

```
52
53
54 // 事件处理函数，epoll_wait()返回的时候，执行它。
55 void Channel::handleevent(Socket *servsock)
56 {
57     if (revents_ & EPOLLRDHUP)           // 对方已关闭，有些系统检测不到，可以使用EPOLLIN，recv()返回0。
58     {
59         printf("client(eventfd=%d) disconnected.\n",fd_);
60         close(fd_);                    // 关闭客户端的fd。
61     }                                // 普通数据 带外数据
62     else if (revents_ & (EPOLLIN|EPOLLPRI)) // 接收缓冲区中有数据可以读。
63     {
64         /*
65         if (islisten_==true) // 如果是listenfd有事件，表示有新的客户端连上来。
66             newconnection(servsock);
67         else                // 如果是客户端连接的fd有事件。
68             onmessage();
69         */
70         readcallback_();                // 调用设置的回调函数。
71     }
72     else if (revents_ & EPOLLOUT)        // 有数据需要写，暂时没有代码，以后再说。
73 }
```

项目文件夹 [www.xitnDS.Com...]

4:05

Channel.h Channel.cpp tcpepoll.cpp

```
netserver > 08 > C Channel.h > Channel.cpp > newconnection(Socket *)
```

```
81
82 // 处理新客户端连接请求。
83 void Channel::newconnection(Socket *servsock)
84 {
85     InetAddress clientaddr;           // 客户端的地址和协议。
86     // 注意, clientsock只能new出来, 不能在栈上, 否则析构函数会关闭fd。
87     // 还有, 这里new出来的对象没有释放, 这个问题以后再解决。
88     Socket *clientsock=new Socket(servsock->accept(clientaddr));
89
90     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
91
92     // 为新客户端连接准备读事件, 并添加到epoll中。
93     Channel *clientchannel=new Channel(ep,clientsock->fd(),false); // 这里new出来的对象没有释放, 这个问题以后再解决。
94     clientchannel->setreadcallback(std::bind(&Channel::onmessage,clientchannel));
95     clientchannel->useet();          // 客户端连上来的fd采用边缘触发。
96     clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
97 }
98
99 // 处理对端发送过来的消息。
100 void Channel::onmessage()
101 {
102     char buffer[1024];
103     while (true)      // 由于使用非阻塞IO, 一次读取buffer大小数据, 直到全部的数据读取完毕。
104     {
105         bzero(&buffer,sizeof(buffer));
```

函数回调 [www.xitnDS.Com...]

4:06

Channel.h Channel.cpp tcpepoll.cpp 1

```
netserver > 08 > C Channel.h > Channel > Channel(Epoll *, int)
```

```
15     bool inepoll_=false;           // Channel是否已添加到epoll树上, 如果未添加, 调用epoll_ctl()的时候用EPOLLIN。
16     uint32_t events_=0;            // fd_需要监视的事件。listenfd和clientfd需要监视EPOLLIN, clientfd还可能需要EPOLLET。
17     uint32_t revents_=0;           // fd_已发生的事件。
18     std::function<void()> readcallback_; // fd_读事件的回调函数。
19
20 public:
21     Channel(Epoll* ep,int fd);    // 构造函数。
22     ~Channel();                  // 析构函数。
23
24     int fd();                     // 返回fd_成员。
25     void useet();                // 采用边缘触发。
26     void enablereading();        // 让epoll_wait()监视fd_的读事件。
27     void setinepoll();           // 把inepoll_成员的值设置为true。
28     void setrevents(uint32_t ev); // 设置revents_成员的值为参数ev。
29     bool inpoll();               // 返回inepoll_成员。
30     uint32_t events();           // 返回events_成员。
31     uint32_t revents();          // 返回revents_成员。
32
33     void handleevent();          // 事件处理函数, epoll_wait()返回的时候, 执行它。
34
35     void newconnection(Socket *servsock); // 处理新客户端连接请求。
36     void onmessage();             // 处理对端发送过来的消息。
37     void setreadcallback(std::function<void()> fn); // 设置fd_读事件的回调函数。
38 }
```

```
netserver > 08 > Channel.cpp > Channel(Channel.h)
1 #include "Channel.h"
2
3 Channel::Channel(Epoll* ep,int fd):ep_(ep),fd_(fd) // 构造函数。
4 {
5
6 }
7
8 Channel::~Channel() // 析构函数。
9 {
10 // 在析构函数中，不要销毁ep_，也不能关闭fd_，因为这两个东西不属于Channel类。
11 }
12
```

```
... Channel.h Channel.cpp tcpepoll.cpp
netserver > 08 > tcpepoll.cpp > main(int, char * [])
150.128] 24 printf("usage: ./tcpepoll ip port\n");
25 printf("example: ./tcpepoll 192.168.150.128 5085\n\n");
26 return -1;
27
28
29 Socket servsock(createnonblocking());
30 InetSocketAddress servaddr(argv[1],atoi(argv[2])); // 服务端的地址和协议。
31 servsock.setreuseaddr(true);
32 servsock.settcpnodelay(true);
33 servsock.setreuseport(true);
34 servsock.setkeepalive(true);
35 servsock.bind(servaddr);
36 servsock.listen();
37
38 Epoll ep;
39 Channel *servchannel=new Channel(&ep,servsock.fd()); // 这里new出来的对象没有释放，这个问题
40 servchannel->setreadcallback(std::bind(&Channel::newconnection,servchannel,&servsock));
41 servchannel->enablereading(); // 让epoll_wait()监视servchannel的读事件。
42
43 while (true) // 事件循环。
44 {
45
46
47
48
49
50
51
52 // 处理新客户端连接请求。
53 void Channel::newconnection(Socket *servsock)
54 {
55 InetSocketAddress clientaddr; // 客户端的地址和协议。
56 // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。
57 // 还有，这里new出来的对象没有释放，这个问题以后再解决。
58 Socket *clientsock=new Socket(servsock->accept(clientaddr));
59
60 printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
61
62 // 为新客户端连接准备读事件，并添加到epoll中。
63 Channel *clientchannel=new Channel(ep_clientsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
64 clientchannel->setreadcallback(std::bind(&Channel::onmessage,clientchannel));
65 clientchannel->useet(); // 客户端连上来的fd采用边缘触发。
66 clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
67 }
68
69 // 处理对端发送过来的消息。
70 void Channel::onmessage()
71 {
```

EventLoop

The screenshot shows a code editor interface with the following details:

- Resource Manager:** Shows the project structure: netserver > 09 > EventLoop.h.
- Open Editors:** tcpepoll.cpp and EventLoop.h.
- Project Structure:** netserver > 01, 02, 03, 04, 05, 06, 07, 08, 09 (selected), Channel.cpp, Channel.h, client_imooe, client.cpp, Epoll.cpp, Epoll.h, EventLoop.h (selected), InetAddress.cpp, InetAddress.h, makefile, Socket.cpp, Socket.h, tcpepoll, tcpepoll.cpp.
- Code Content:**

```
1 #pragma once
2 #include "Epoll.h"
3
4 // 事件循环类。
5 class EventLoop
6 {
7 private:
8     Epoll *ep_; // 每个事件循环只有一个Epoll。
9 public:
10    EventLoop(); // 在构造函数中创建Epoll对象ep_。
11    ~EventLoop(); // 在析构函数中销毁ep_。
12
13    void run(); // 运行事件循环。
14 }
```

```
tcpepoll.cpp EventLoop.h EventLoop.cpp X
netserver > 09 > EventLoop.cpp > run()
15 */
16
17 I
18 // 在构造函数中创建Epoll对象ep_。
19 EventLoop::EventLoop():ep_(new Epoll)
20 {
21
22 }
23
24 // 在析构函数中销毁ep_。
25 EventLoop::~EventLoop()
26 {
27     delete ep_;
28 }
29
30 // 运行事件循环。
31 void EventLoop::run()
32 {
33     while (true)      // 事件循环。
34     {
35         std::vector<Channel *> channels=ep_->loop();    // 等待监视的fd有事件发生。
36
37         for (auto &ch:channels)
38         {
39             ch->handleevent();    // 处理epoll_wait()返回的事件。
40         }
41     }
42 }
```

```
.. tcpepoll.cpp 1 EventLoop.h X EventLoop.cpp
netserver > 09 > EventLoop.h > EventLoop > ep()
28]
1 #pragma once
2 #include "Epoll.h"
3
4 // 事件循环类。
5 class EventLoop
6 {
7 private:
8     Epoll *ep_;           // 每个事件循环只有一个Epoll。
9 public:
10    EventLoop();          // 在构造函数中创建Epoll对象ep_。
11    ~EventLoop();         // 在析构函数中销毁ep_。
12
13    void run();           // 运行事件循环。
14    Epoll *ep[0];         I
15};
```

EventLoop类 [WWW.XiTnDS.COM...]

5:01

截图提取文字 帮我写观后感 视频的主要

tcp epoll.cpp 1 EventLoop.h EventLoop.cpp X

netserver > 09 > EventLoop.cpp > ep0

```
28] 22 }
23
24 // 在析构函数中销毁ep_。
25 EventLoop::~EventLoop()
26 {
27     delete ep_;
28 }
29
30 // 运行事件循环。
31 void EventLoop::run()
32 {
33     while (true)      // 事件循环。
34     {
35         std::vector<Channel *> channels=ep_->loop();    // 等待监视的fd有事件发生。
36
37         for (auto &ch:channels)
38         {
39             ch->handleevent();    // 处理epoll_wait()返回的事件。
40         }
41     }
42 }
43
44 Epoll* EventLoop::ep()
45 {
46     return ep_;
47 }
```

TcpServer

tcp epoll.cpp TcpServer.h ...

netserver > 11 > TcpServer.h > ...

```
1 #pragma once
2 #include "EventLoop.h"
3 #include "Socket.h"
4 #include "Channel.h"
5
6 class TcpServer
7 {
8 private:
9     EventLoop loop_; // 一个TcpServer可以有多个事件循环，现在是单线程，暂时只用一个事件循环。
10 public:
11     TcpServer(const std::string &ip,const uint16_t port);
12     ~TcpServer();
13 }
```

```
C TcpServer.h  C TcpServer.cpp X  C Socket.cpp
netserver > 11 > C TcpServer.cpp > T TcpServer(const std::string &, const uint16_t)
10 | ~TcpServer();
11 };*/
12
13 TcpServer::TcpServer(const std::string &ip,const uint16_t port)
14 {
15     Socket *servsock=new Socket(creatnonblocking()); // 这里new出来的对象没有释放，以后再说。
16     InetAddress servaddr(ip,port);           // 服务端的地址和协议。
17     servsock->setreuseaddr(true);
18     servsock->settcpnodelay(true);
19     servsock->setreuseport(true);
20     servsock->setkeepalive(true);
21     servsock->bind(servaddr);
22     servsock->listen();
23
24     EventLoop loop;
25     Channel *servchannel=new Channel(&loop,servsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
26     servchannel->setreadcallback(std::bind(&Channel::newconnection,servchannel,servsock));
27     servchannel->enablereading(); // 让epoll_wait()监视servchannel的读事件。
28 }
29
30 TcpServer::~TcpServer()
31 {
32 }
```

```
project [SSH: 192.168.150.128]

C TcpServer.h X C+ TcpServer.cpp C+ tcpepoll.cpp 2

netserver > 11 > C TcpServer.h > TcpServer
1 #pragma once
2 #include "EventLoop.h"
3 #include "Socket.h"
4 #include "Channel.h"
5
6 class TcpServer
7 {
8 private:
9     EventLoop loop_; // 一个TcpServer可以有多个事件循环，现在是单线程，暂时只用一个事件循环。
10 public:
11     TcpServer(const std::string &ip,const uint16_t port);
12     ~TcpServer();
13
14     void start(); // 运行事件循环。
15 };
```

```
C TcpServer.h X C+ TcpServer.cpp C+ tcpepoll.cpp 2

netserver > 11 > C TcpServer.cpp > start()
18 servsock->settcpnodelay(true);
19 servsock->setreuseport(true);
20 servsock->setkeepalive(true);
21 servsock->bind(servaddr);
22 servsock->listen();
23
24 EventLoop loop;
25 Channel *servchannel=new Channel(&loop,servsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
26 servchannel->setreadcallback(std::bind(&Channel::newconnection,servchannel,servsock));
27 servchannel->enablereading(); // 让epoll_wait()监视servchannel的读事件。
28 }
29
30 TcpServer::~TcpServer()
31 {
32
33 }
34
35 // 运行事件循环。
36 void TcpServer::start()
37 {
38     loop_.run();
39 }
```

```
● TcpServer.h ● TcpServer.cpp ● tcpepoll.cpp
netserver > 11 > TcpServer.cpp > TcpServer(const std::string &, const uint16_t)
11 } */
12
13 TcpServer::TcpServer(const std::string &ip,const uint16_t port)
14 {
15     Socket *servsock=new Socket(createNonblocking()); // 这里new出来的对象没有释放，以后再说。
16     InetSocketAddress servaddr(ip,port);           // 服务端的地址和协议。
17     servsock->setReuseAddr(true);
18     servsock->setTcpNoDelay(true);
19     servsock->setReusePort(true);
20     servsock->setKeepAlive(true);
21     servsock->bind(servaddr);
22     servsock->listen();
23
24     Channel *servchannel=new Channel(&loop,servsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
25     servchannel->setReadCallback(std::bind(&Channel::newConnection,servchannel,servsock));
26     servchannel->enableReading(); // 让epoll_wait()监视servchannel的读事件。
27 }
28
29 TcpServer::~TcpServer()
30 {
31
32 }
33
34 // 运行事件循环。
35 void TcpServer::start()
36 {
37     loop.run();

```

```
● TcpServer.h ● TcpServer.cpp ● Acceptor.h X
netserver > 12 > Acceptor.h > Acceptor > Acceptor(EventLoop *, const std::string &, const uint16_t)
1 #pragma once
2 #include <functional>
3 #include "Socket.h"
4 #include "InetAddress.h"
5 #include "Channel.h"
6 #include "EventLoop.h"
7
8 class Acceptor
9 {
10 private:
11     EventLoop *loop_;          // Acceptor对应的事件循环，在构造函数中传入。
12     Socket *servsock_;         // 服务端用于监听的socket，在构造函数中创建。
13     Channel *acceptchannel_; // Acceptor对应的channel，在构造函数中创建。
14 public:
15     Acceptor(EventLoop *loop,const std::string &ip,const uint16_t port);
16     ~Acceptor();
17 };
18
```

```
C TcpServer.h ● C TcpServer.cpp Acceptor.h C Acceptor.cpp X
netserver > 12 > C Acceptor.cpp > ~Acceptor()
1 #include "Acceptor.h"
2
3 Acceptor::Acceptor(EventLoop *loop,const std::string &ip,const uint16_t port):loop_(loop)
4 {
5     servsock_=new Socket(createonblocking());
6     InetAddress servaddr(ip,port);          // 服务端的地址和协议。
7     servsock_->setreuseaddr(true);
8     servsock_->settcpnodelay(true);
9     servsock_->setreuseport(true);
10    servsock_->setkeepalive(true);
11    servsock_->bind(servaddr);
12    servsock_->listen();
13
14    acceptchannel_=new Channel(loop_,servsock_->fd());
15    acceptchannel_->setreadcallback(std::bind(&Channel::newconnection,acceptchannel_,servsock_));
16    acceptchannel_->enablereading();      // 让epoll_wait()监视servchannel的读事件。
17 }
18
19 Acceptor::~Acceptor()
20 {
21     delete servsock_;
22     delete acceptchannel_;
23 }
```

```
C TcpServer.h C TcpServer.cpp X C Acceptor.h C Acceptor.cpp
netserver > 12 > C TcpServer.cpp > ~TcpServer()
1
2     Socket *servsock=new Socket(createonblocking()); // 这里new出来的对象没有释放，以后再说。
3     InetAddress servaddr(ip,port);          I // 服务端的地址和协议。
4     servsock->setreuseaddr(true);
5     servsock->settcpnodelay(true);
6     servsock->setreuseport(true);
7     servsock->setkeepalive(true);
8     servsock->bind(servaddr);
9     servsock->listen();
10
11     Channel *servchannel=new Channel(&loop_,servsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
12     servchannel->setreadcallback(std::bind(&Channel::newconnection,servchannel,servsock));
13     servchannel->enablereading(); // 让epoll_wait()监视servchannel的读事件。
14 */
15     acceptor_=new Acceptor(&loop_,ip,port);
16 }
17
18 TcpServer::~TcpServer()
19 {
20     delete acceptor_;
21 }
22
23 // 运行事件循环。
24 void TcpServer::start()
25 {
26     loop_.run();
27 }
```

connection

Connection.h ● Channel.cpp

```
netserver > 13 > Connection.h > Connection > Connection(EventLoop *, Socket *)
1 #pragma once
2 #include <functional>
3 #include "Socket.h"
4 #include "InetAddress.h"
5 #include "Channel.h"
6 #include "EventLoop.h"
7
8 class Connection
9 {
10 private:
11     EventLoop *loop_;           // Connection对应的事件循环，在构造函数中传入。
12     Socket *clientsock_;       // 与客户端通讯的Socket。
13     Channel *clientchannel_;   // Connection对应的channel，在构造函数中创建。
14 public:
15     Connection(EventLoop *loop, Socket *clientsock);
16     ~Connection();
17 };
18
```

Connection类『免费资源_xiTndS.co...』 5:15

Connection.h Connection.cpp X Channel.cpp

```
netserver > 13 > Connection.cpp > ~Connection()
1 #include "Connection.h"
2
3
4 Connection::Connection(EventLoop *loop, Socket *clientsock) : loop_(loop), clientsock_(clientsock)
5 {
6     // 为新客户端连接准备读事件，并添加到epoll中。
7     clientchannel_ = new Channel(loop_, clientsock_->fd());
8     clientchannel_->setreadcallback(std::bind(&Channel::onmessage, clientchannel_));
9     clientchannel_->useet();           // 客户端连上的fd采用边缘触发。
10    clientchannel_->enablereading(); // 让epoll_wait()监视clientchannel的读事件
11 }
12
13 Connection::~Connection()
14 {
15     delete clientsock_;
16     delete clientchannel_;
17 }
```

```
... C Connection.h C Connection.cpp C Channel.cpp ●
netserver > 13 > C Channel.cpp > newconnection(Socket *)
28 }           // 从新连接的事件中移除。
72     close(fd_); // 关闭客户端的fd。
73 }
74 }
75
76 #include "Connection.h"
77
78 // 处理新客户端连接请求。
79 void Channel::newconnection(Socket *servsock)
80 {
81     InetSocketAddress clientaddr; // 客户端的地址和协议。
82     // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。
83     // 还有，这里new出来的对象没有释放，这个问题以后再解决。
84     Socket *clientsock=new Socket(servsock->accept(clientaddr));
85
86     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
87
88     /*
89     // 为新客户端连接准备读事件，并添加到epoll中。
90     Channel *clientchannel=new Channel(loop_,clientsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
91     clientchannel->setreadcallback(std::bind(&Channel::onmessage,clientchannel));
92     clientchannel->useet(); // 客户端连上来的fd采用边缘触发。
93     clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
94     */
95     Connection *conn=new Connection(loop_,clientsock); // 这里new出来的对象没有释放，这个问题以后再解决。
96 }
```

优化回调函数

```
Acceptor类的成员函数『海量免费资...
project [SSH: 192.168.150.128] 5:16
C Channel.h ●
netserver > 14 > C Channel.h > Channel
8 }           // Channel类的成员函数
15     bool inepoll_=false; // Channel是否已添加到epoll树上，如果未添加，调用epoll_ctl()的时候用EPOLL_CTL_ADD，否则用EPOLL_CTL_MOD。
16     uint32_t events_=0; // fd 需要监视的事件。listenfd和clientfd需要监视EPOLLIN，clientfd还可能需要监视EPOLLOUT。
17     uint32_t revents_=0; // fd_已发生的事件。
18     std::function<void()> readcallback_; // fd_读事件的回调函数。
19
20 public:
21     Channel(EventLoop* loop,int fd); // 构造函数。 Channel是Acceptor和Connection的下层类。
22     ~Channel(); // 析构函数。
23
24     int fd(); // 返回fd_成员。
25     void useet(); // 采用边缘触发。
26     void enablereading(); // 让epoll_wait()监视fd_的读事件。
27     void setinepoll(); // 把inepoll_成员的值设置为true。
28     void setrevents(uint32_t ev); // 设置revents_成员的值为参数ev。
29     bool inpoll(); // 返回inepoll_成员。
30     uint32_t events(); // 返回events_成员。
31     uint32_t revents(); // 返回revents_成员。
32
33     void handleevent(); // 事件处理函数，epoll_wait()返回的时候，执行它。
34
35     void newconnection(Socket *servsock); // 处理新客户端连接请求。
36     void onmessage(); // 处理对端发送过来的消息。
37     void setreadcallback(std::function<void()> fn); // 设置fd_读事件的回调函数。
38 };
```

```
文件树|最近更改|历史记录|全局搜索...          窗口          帮助
Channel.h ●  C Acceptor.h ●  C Acceptor.cpp 2 ×
netserver > 14 > C Acceptor.cpp > newconnection(Socket *)
17 }
18
19 Acceptor::~Acceptor()
20 {
21     delete servsock_;
22     delete acceptchannel_;
23 }
24
25 #include "Connection.h"
26
27 // 处理新客户端连接请求。
28 void Acceptor::newconnection(Socket *servsock)
29 {
30     InetAddress clientaddr;           // 客户端的地址和协议。
31     // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。
32     // 还有，这里new出来的对象没有释放，将在Connection类的析构函数中释放。
33     Socket *clientsock=new Socket(servsock->accept(clientaddr));
34
35     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
36
37     /*
38     // 为新客户端连接准备读事件，并添加到epoll中。
39     Channel *clientchannel=new Channel(loop_,clientsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
40     clientchannel->setreadcallback(std::bind(&Channel::onmessage,clientchannel));
41     clientchannel->useet();           // 客户端连上来的fd采用边缘触发。
42     clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
43     */
44     Connection *conn=new Connection(loop_,clientsock); // 这里new出来的对象没有释放，这个问题以后再解决。
45 }
```

```
File|最近更改|历史记录|全局搜索...          窗口          帮助
Channel.h X  C Acceptor.h ●  C Acceptor.cpp 2
netserver > 14 > C Channel.h > Channel
15     bool inepoll_=false;           // Channel是否已添加到epoll树上，如果未添加，调用epoll_ctl()的时候用EPOLL_CTL_ADD，否
16     uint32_t events_=0;           // fd_需要监视的事件。listenfd和clientfd需要监视EPOLLIN，clientfd还可能需要监视EPOLLOUT
17     uint32_t revents_=0;           // fd_已发生的事件。
18     std::function<void()> readcallback_; // fd_读事件的回调函数。
19
20 public:
21     Channel(EventLoop* loop,int fd); // 构造函数。                               Channel是Acceptor和Connection的下层类。
22     ~Channel();                   // 析构函数。
23
24     int fd();                     // 返回fd_成员。
25     void useet();                // 采用边缘触发。
26     void enablereading();        // 让epoll_wait()监视fd_的读事件。
27     void setinepoll();           // 把inepoll_成员的值设置为true。
28     void setrevents(uint32_t ev); // 设置revents_成员的值为参数ev。
29     bool inpoll();               // 返回inepoll_成员。
30     uint32_t events();           // 返回events_成员。
31     uint32_t revents();          // 返回revents_成员。
32
33     void handleevent();          // 事件处理函数，epoll_wait()返回的时候，执行它。
34
35     //void newconnection(Socket *servsock); // 处理新客户端连接请求。
36     void onmessage();            // 处理对端发送过来的消息。
37     void setreadcallback(std::function<void()> fn); // 设置fd_读事件的回调函数。
38 };
```

Acceptor类的成员函数『海量免费资源』

Channel.h Channel.cpp Acceptor.h Acceptor.cpp 2

```
0.128] 72     close(fd); // 关闭客户端的fd。
73 }
74 }
75
76 /*
77 #include "Connection.h"
78
79 // 处理新客户端连接请求。
80 void Channel::newconnection(Socket *servsock)
81 {
82     InetSocketAddress clientaddr; // 客户端的地址和协议。
83     // 注意, clientsock只能new出来, 不能在线上, 否则析构函数会关闭fd。
84     // 还有, 这里new出来的对象没有释放, 将在Connection类的析构函数中释放。
85     Socket *clientsock=new Socket(servsock->accept(clientaddr));
86
87     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
88
89     Connection *conn=new Connection(loop_,clientsock); // 这里new出来的对象没有释放, 这个问题以后再解决。
90 }
91 */
```

Acceptor类的成员函数『海量免费资源』

5:19

Channel.h Channel.cpp Acceptor.h Acceptor.cpp

```
netserver > 14 > Acceptor.cpp > Acceptor(EventLoop *, const std::string &, const uint16_t)
1 #include "Acceptor.h"
2
3 Acceptor::Acceptor(EventLoop *loop,const std::string &ip,const uint16_t port):loop_(loop)
4 {
5     servsock_=new Socket(createonblocking());
6     InetSocketAddress servaddr(ip,port); // 服务端的地址和协议。
7     servsock_->setreuseaddr(true);
8     servsock_->settcpnodelay(true);
9     servsock_->setreuseport(true);
10    servsock_->setkeepalive(true);
11    servsock_->bind(servaddr);
12    servsock_->listen();
13
14    acceptchannel_=new Channel(loop_,servsock_->fd());
15    // acceptchannel_->setreadcallback(std::bind(&Channel::newconnection,acceptchannel_,servsock_));
16    acceptchannel_->setreadcallback(std::bind(&Acceptor::newconnection,this,servsock_));
17    acceptchannel_->enablereading(); // 让epoll_wait()监视servchannel的读事件。
18 }
```

```
Channel.h Channel.cpp Acceptor.h Acceptor.cpp
netserver > 14 > C Acceptor.h > Acceptor > newconnection()
1 #pragma once
2 #include <functional>
3 #include "Socket.h"
4 #include "InetAddress.h"
5 #include "Channel.h"
6 #include "EventLoop.h"
7
8 class Acceptor
9 {
10 private:
11     EventLoop *loop_;           // Acceptor对应的事件循环，在构造函数中传入。
12     Socket *servsock_;         // 服务端用于监听的socket，在构造函数中创建。
13     Channel *acceptchannel_;   // Acceptor对应的channel，在构造函数中创建。
14 public:
15     Acceptor(EventLoop *loop,const std::string &ip,const uint16_t port);
16     ~Acceptor();
17
18     void newconnection();      // 处理新客户端连接请求。
19 };
20
```

```
Channel.h Channel.cpp Acceptor.h Acceptor.cpp 2
netserver > 14 > C Acceptor.cpp > newconnection()
21 {
22     delete servsock_;
23     delete acceptchannel_;
24 }
25
26 #include "Connection.h"
27
28 // 处理新客户端连接请求。
29 void Acceptor::newconnection()
30 {
31     InetAddress clientaddr;    // 客户端的地址和协议。
32     // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。
33     // 还有，这里new出来的对象没有释放，将在Connection类的析构函数中释放。
34     Socket *clientsock=new Socket(servsock_->accept(clientaddr));
35
36     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
37
38     /*
39     // 为新客户端连接准备读事件，并添加到epoll中。
40     Channel *clientchannel=new Channel(loop_,clientsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
41     clientchannel->setreadcallback(std::bind(&Channel::onmessage,clientchannel));
42     clientchannel->useet();           // 客户端连上的fd采用边缘触发。
43     clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
44     */
45     Connection *conn=new Connection(loop_,clientsock); // 这里new出来的对象没有释放，这个问题以后再解决。
46 }
```

or类的成员函数 海量免费资...

5.21

截图提取文字 帮我与观后感 演示视图

Channel.h Channel.cpp Acceptor.h Acceptor.cpp

```
netserver > 14 > Acceptor.cpp > Acceptor(EventLoop *, const std::string &, const uint16_t)
12 servsock_ ->listen();
13
14 acceptchannel_=new Channel(loop_,servsock_->fd());
15 // acceptchannel_->setreadcallback(std::bind(&Channel::newconnection,acceptchannel_,servsock_));
16 acceptchannel_->setreadcallback(std::bind(&Acceptor::newconnection,this));
17 acceptchannel_->enablereading(); // 让epoll_wait()监视servchannel的读事件。
18 }
19
20 Acceptor::~Acceptor()
21 {
22     delete servsock_;
23     delete acceptchannel_;
24 }
25
26 #include "Connection.h"
27
28 // 处理新客户端连接请求。
29 void Acceptor::newconnection()
30 {
31     InetAddress clientaddr; // 客户端的地址和协议。
32     // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。
33     // 还有，这里new出来的对象没有释放，将在Connection类的析构函数中释放。
34 }
```

Acceptor.h TcpServer.h TcpServer.cpp Acceptor.cpp

```
netserver > 15 > TcpServer.h > TcpServer
1 #pragma once
2 #include "EventLoop.h"
3 #include "Socket.h"
4 #include "Channel.h"
5 #include "Acceptor.h"
6
7 // TCP网络服务类。
8 class TcpServer
9 {
10 private:
11     EventLoop loop_; // 一个TcpServer可以有多个事件循环，现在是单线程，暂时只用一个事件循环。
12     Acceptor *acceptor_; // 一个TcpServer只有一个Acceptor对象。
13 public:
14     TcpServer(const std::string &ip,const uint16_t port);
15     ~TcpServer();
16
17     void start(); // 运行事件循环。
18
19     void newconnection(); // 处理新客户端连接请求。
20 };
```

Server类的成员函数『免费资源...』 5:24 截图提取文字 帮我写观后

C Acceptor.h C TcpServer.h ● C TcpServer.cpp 1 C Acceptor.cpp

```
netserver > 15 > C TcpServer.h > newconnection(Socket *)
```

```
1 #pragma once
2 #include "EventLoop.h"
3 #include "Socket.h"
4 #include "Channel.h"
5 #include "Acceptor.h"
6 #include "Connection.h"
7
8 // TCP网络服务类。
9 class TcpServer
10 {
11 private:
12     EventLoop loop_; // 一个TcpServer可以有多个事件循环，现在是单线程，暂时只用一个事件循环。
13     Acceptor *acceptor_; // 一个TcpServer只有一个Acceptor对象。
14 public:
15     TcpServer(const std::string &ip, const uint16_t port);
16     ~TcpServer();
17
18     void start(); // 运行事件循环。
19
20     void newconnection(Socket *clientsock); // 处理新客户端连接请求。
21 };
```

C Acceptor.h C TcpServer.h C TcpServer.cpp X C Acceptor.cpp

```
netserver > 15 > C TcpServer.cpp > newconnection(Socket *)
```

```
21
22     TcpServer::~TcpServer()
23     {
24         delete acceptor_;
25     }
26
27     // 运行事件循环。
28     void TcpServer::start()
29     {
30         loop_.run();
31     }
32
33     // 处理新客户端连接请求。
34     void TcpServer::newconnection(Socket* clientsock)
35     {
36         Connection *conn=new Connection(&loop_,clientsock); // 这里new出来的对象没有释放，这个问题以后再解决。
37     }
38 }
```

Acceptor类的成员函数『免费资源...』

Acceptor.h TcpServer.h TcpServer.cpp Acceptor.cpp

```
netserver > 15 > C: Acceptor.cpp > newconnection()
23     delete acceptchannel_;
24 }
25
26 #include "Connection.h"
27
28 // 处理新客户端连接请求。
29 void Acceptor::newconnection()
30 {
31     InetSocketAddress clientaddr; // 客户端的地址和协议。
32     // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。
33     // 还有，这里new出来的对象没有释放，将在Connection类的析构函数中释放。
34     Socket *clientsock=new Socket(servsock_->accept(clientaddr));
35
36     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
37
38     /*
39     // 为新客户端连接准备读事件，并添加到epoll中。
40     Channel *clientchannel=new Channel(loop_,clientsock->fd()); // 这里new出来的对象没有释放，这个问题以后再解决。
41     clientchannel->setreadcallback(std::bind(&Channel::onmessage,clientchannel));
42     clientchannel->useet(); // 客户端连上的fd采用边缘触发。
43     clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
44     */
45     // Connection conn=new Connection(loop_,clientsock); // 这里new出来的对象没有释放，这个问题以后再解决。
46 }
47
```

Acceptor.h TcpServer.h TcpServer.cpp Acceptor.cpp

```
netserver > 15 > C: Acceptor.h > Acceptor
1 #pragma once
2 #include <functional>
3 #include "Socket.h"
4 #include "InetAddress.h"
5 #include "Channel.h"
6 #include "EventLoop.h"
7
8 class Acceptor
9 {
10 private:
11     EventLoop *loop_; // Acceptor对应的事件循环，在构造函数中传入。
12     Socket *servsock_; // 服务端用于监听的socket，在构造函数中创建。
13     Channel *acceptchannel_; // Acceptor对应的channel，在构造函数中创建。
14     std::function<void(Socket*)> newconnectioncb_; // 处理新客户端连接请求的回调函数，将指向TcpServer::newconnection()
15
16 public:
17     Acceptor(EventLoop *loop,const std::string &ip,const uint16_t port);
18     ~Acceptor();
19
20     void newconnection(); // 处理新客户端连接请求。
21
22     void setnewconnectioncb(std::function<void(Socket*)> fn); // 设置处理新客户端连接请求的回调函数。
23 };
24
```

```
Acceptor.h   C TcpServer.h   C+ TcpServer.cpp   C+ Acceptor.cpp 2 ●
netserver > 15 > C Acceptor.cpp > ...
32 // 注意, clientsock只能new出来, 不能在栈上, 否则析构函数会关闭fd。
33 // 还有, 这里new出来的对象没有释放, 将在Connection类的析构函数中释放。
34 Socket *clientsock=new Socket(servsock_->accept(clientaddr));
35
36 printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
37
38 /*
39 // 为新客户端连接准备读事件, 并添加到epoll中。
40 Channel *clientchannel=new Channel(loop_,clientsock->fd()); // 这里new出来的对象没有释放, 这个问题以后再解决。
41 clientchannel->setreadcallback(std::bind(&Channel::onmessage,clientchannel));
42 clientchannel->useet(); // 客户端连上来的fd采用边缘触发。
43 clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
44 */
45 // Connection *conn=new Connection(loop_,clientsock); // 这里new出来的对象没有释放, 这个问题以后再解决。
46 }
47
48 void Acceptor::setnewconnectioncb(std::function<void(Socket*)> fn); // 设置处理新客户端连接请求的回调函数。
49 {
50     newconnectioncb_=fn;
51 }
52
```

```
Acceptor.h   C TcpServer.h   C+ TcpServer.cpp   C+ Acceptor.cpp X
netserver > 15 > C Acceptor.cpp > newconnection()
28 // 处理新客户端连接请求。
29 void Acceptor::newconnection()
30 {
31     InetSocketAddress clientaddr; // 客户端的地址和协议。
32     // 注意, clientsock只能new出来, 不能在栈上, 否则析构函数会关闭fd。
33     // 还有, 这里new出来的对象没有释放, 将在Connection类的析构函数中释放。
34     Socket *clientsock=new Socket(servsock_->accept(clientaddr));
35
36     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());
37
38 /*
39 // 为新客户端连接准备读事件, 并添加到epoll中。
40 Channel *clientchannel=new Channel(loop_,clientsock->fd()); // 这里new出来的对象没有释放, 这个问题以后再解决。
41 clientchannel->setreadcallback(std::bind(&Channel::onmessage,clientchannel));
42 clientchannel->useet(); // 客户端连上来的fd采用边缘触发。
43 clientchannel->enablereading(); // 让epoll_wait()监视clientchannel的读事件
44 */
45 // Connection *conn=new Connection(loop_,clientsock); // 这里new出来的对象没有释放, 这个问题以后再解决。
46     newconnectioncb_(clientsock);
47 }
48 void Acceptor::setnewconnectioncb(std::function<void (Socket *)> fn)
49 void Acceptor::setnewconnectioncb(std::function<void(Socket*)> fn) // 设置处理新客户端连接请求的回调函数。
50 {
51     newconnectioncb_=fn;
52 }
```

```
C Acceptor.h C TcpServer.h C+ TcpServer.cpp X C+ Acceptor.cpp  
netserver > 15 > C+ TcpServer.cpp > TcpServer(const std::string &, const uint16_t)  
3 | tcpserver::tcpserver(const std::string &ip, const uint16_t port)  
4 | {  
5 |     acceptor_=new Acceptor(&loop_,ip,port);  
6 |     acceptor_->setnewconnectioncb(std::bind(&TcpServer::newconnection,this,std::placeholders::_1));  
7 | }  
8 |  
9 | TcpServer::~TcpServer()  
10 | {  
11 |     delete acceptor_;  
12 | }  
13 |  
14 | // 运行事件循环。  
15 | void TcpServer::start()  
16 | {  
17 |     loop_.run();  
18 | }  
19 |  
20 | // 处理新客户端连接请求。  
21 | void TcpServer::newconnection(Socket* clientsock)  
22 | {  
23 |     Connection *conn=new Connection(&loop_,clientsock); // 这里new出来的对象没有释放，这个问题以后再解决。  
24 | }  
25 |
```

```
C Acceptor.h C TcpServer.h C+ TcpServer.cpp X C+ Acceptor.cpp X  
netserver > 15 > C+ Acceptor.cpp > newconnection()  
23 |     delete acceptchannel_;  
24 | }  
25 |  
26 | #include "Connection.h"  
27 |  
28 | // 处理新客户端连接请求。  
29 | void Acceptor::newconnection()  
30 | {  
31 |     InetAddress clientaddr; // 客户端的地址和协议。  
32 |     // 注意，clientsock只能new出来，不能在栈上，否则析构函数会关闭fd。  
33 |     // 还有，这里new出来的对象没有释放，将在Connection类的析构函数中释放。  
34 |     Socket *clientsock=new Socket(servsock_->accept(clientaddr));  
35 |  
36 |     printf ("accept client(fd=%d,ip=%s,port=%d) ok.\n",clientsock->fd(),clientaddr.ip(),clientaddr.port());  
37 |  
38 |     // Connection *conn=new Connection(loop_,clientsock); // 这里new出来的对象没有释放，这个问题以后再解决。  
39 |     newconnectioncb_(clientsock);  
40 | }  
41 |  
42 | void Acceptor::setnewconnectioncb(std::function<void(Socket*)> fn) // 设置处理新客户端连接请求的回调函数。  
43 | {  
44 |     newconnectioncb_=fn;  
45 | }
```